10-2018

Function risk assessment under memory leakage

Jianming FU
Wuhan University

Rui JIN
Wuhan University

Yan LIN
Singapore Management University, yanlin.2016@phdis.smu.edu.sg

Baihe JIANG
Wuhan University

Zhengwei GUO
Wuhan University

Published in 2018 International Conference on Networking and Network Applications: NaNA: Xi'an, China, October 12-15: Proceedings. 284
DOI: 10.1109/NANA.2018.8648754

Function Risk Assessment under Memory Leakage

Jianming Fu
School of Cyber Science and Engineering
Wuhan University
Hubei, China
jmfu@whu.edu.cn

Rui Jin
School of Cyber Science and Engineering
Wuhan University
Hubei, China
jr2016@whu.edu.cn

Yan Lin
School of Information Systems
Singapore Management University
Singapore
yanlin.2016@phdis.smu.edu.sg

Baihe Jiang
School of Cyber Science and Engineering
Wuhan University
Hubei, China
bhjiang.whu@hotmail.com

Zhengwei Guo
School of Cyber Science and Engineering
Wuhan University
Hubei, China
zhengweiguo@whu.edu.cn

***Abstract*—Code reuse attack (CRA), specifically reusing and then reconstructing the codes (gadgets) already existed in programs and libraries, is widely exploited in software attacks. Admittedly, determination of the location of the gadgets consisted of target instructions along with control flow transfer instructions, is of critical importance. Address Space Randomization (ASR), which serves as an effective technique to mitigate CRA, increases the entropy by randomizing the location of the code or data, and baffles adversaries from figuring out the memory layout. Currently, variable randomization methods of high granularity are proposed by scholars to prevent adversaries from deducting memory layout. However, their credibility on alleviating CRA is yet to be confirmed, especially when the suitable pointer is exposed to adversaries. In this paper, we focus on studying what kinds of function leakage can lead to a CRA more likely. A function risk assessment model focusing on function coupling is proposed to quantify the risk caused by the suitable function pointer leakage and it is extended to assess the risk of the whole program and library under the memory leakage. Our experimental results show that popular open-source software is vulnerable when certain code pointer is leaked to adversaries and even severer when the system library is accessible. In addition, suggestions to eliminate function coupling and evaluate the availability of randomization methods are further discussed.**

***Keywords*-Code Reuse Attack; Randomization; Risk Assessment; Memory Leakage;**

I. Introduction

The memory war is effectively an arms race between offense and defense [1]. Code injection is one of the classic attacks by transferring control flow to the injected shellcode. However, Data execution prevention (DEP) [2] prevents code injection by restricting the execution of data segments. To bypass the non-executable data policy, attackers start to use Code Reuse Attack (CRA), which hijacks the control flow of programs by reusing several existing code (gadgets) and has been proven to be Turing complete [3].

Since CRA needs to obtain the accurate addresses of the used gadgets, Address Space Randomization(ASR) is proposed by remapping the memory layout to make it more difficult for attackers to guess the memory layout as their offline analysis are not accurate. Address space layout randomization (ASLR) [4], belonging to ASR, randomizes the base address of the memory segment, has been widely used in operating system. However, attackers can use information leakage to get a memory address, and then guess the memory layout according to this address. Function-level randomization is also one kind of ASR, which randomizes the order of the function in memory to achieve finer granularity of the randomization. Compared with ASLR, it increases the entropy of randomization and makes offline analysis more difficult. But in the case of information leakage, its entropy does not seem to be able to measure its effectiveness. This is also true of fine-grained randomization.

When the memory address of a function is leaked, the attacker can traverse the internal information of the function through the program vulnerability. A function often includes arithmetic instructions, stack pivot instructions, stack pop/push instructions, and control transfer instructions (such as direct or indirect function calls, function return instructions, and direct or indirect jump instructions). An attacker can find the location of other functions by reading the function calls. The gadgets chain in CRA can be constructed by using the return instruction, the indirect jump instruction, and the indirect function call instruction. One issue here is up for debate: what kinds of functions make it easy for an attacker to finish the attack under information leakage? It is necessary to explore the function risk under memory leakage. In this paper, we propose a function risk metric model which can be exploited by both adversaries and defenders. It measures the function risk focusing on the prudent attacker, and it is extended to measure the overall program.

In summary, this paper makes the following contributions:

- We establish an attack model under information leakage

of a pointer which can bypass randomization methods.
- We establish a function risk assessment model focusing on function coupling, which memory permission modification functions, indirect function instructions and system calls are taken into consideration;
- We make an empirical study of the attack model and the assessment model and give some suggestions to eliminate function coupling and evaluate the availability of randomization methods.

This paper is organized as follows: Section II describes the background of this paper including CRA, randomization and information leakage. Section III sets up the model of attacks. Security metric of function coupling risk is discussed in section IV. Our experimental results are shown in section V. Section VI discusses the advantage and limitation of the model and gives suggestions for future randomization. Finally, related works and conclusions are illustrated in section VII and VIII.

II. Background

We first review the basics of important concepts (namely, the code reuse attack strategy and fine-grained memory and code randomization) that are vital to understanding the remainder of this paper.

A. Code Reuse Attack

Code Reuse Attack (CRA) hijacks the control flow of a program by reusing several existing code (gadgets) without injecting external code into memory and has been proven to be Turing complete. Adversaries can use CRA and code injection to bypass the security mechanism, such as ASLR and DEP. For example, if an attacker wants to execute some code with the protection of DEP, the first step is to chain some gadgets to disable DEP. Without DEP protection, EIP is overwritten so as to point to the beginning of the shellcode which is injected before, then the shellcode will be successfully executed.

To accomplish a CRA, one of the challenges concerning the attacks' solution is to ensure the exact address of each gadget in the memory space. Since ASLR randomizes the base address of the target program, it makes the attacker unable to obtain these addresses off-line without information leakage vulnerability or brute force.

B. Randomization

With CRA applied widely, security researchers have proposed a variety of randomization strategies. The main idea of which is to randomize the position of the target program's code and data segments in the memory space to increase the difficulty of gadget chain execution as expected. Randomization techniques are classified into four main categories: Base address randomization [4], [5], Function-level randomization [6], [7], [8], [9], Basic block-level randomization [10], [11], [12]and Instruction-level randomization [13], [14], [15], [16]. Those randomization techniques are listed in table I.

Table I
DIFFERENT RANDOMIZATION TECHNIQUES.

Randomization Techniques	Description
Base address Randomization	Randomize base address
Function-level Randomization	Randomize function locations
Basic-block-level Randomization	Randomize basic block locations
Instruction-level Randomization	Randomize instruction locations or replace instruction sequence

ASLR, attached to base address randomization, randomizes the base address of stack, heap, code segment and libraries. Computer programs written by the high-level languages such as C/C++ are mostly composed of functions. Function-level randomizations reorder the location of functions, and thus the addresses of gadgets are shifted by the randomizations, which baffle adversaries from figuring out the memory layout. Function-level randomization is mainly designed against the off-line analysis from where adversaries obtain gadgets for the CRA. Other fine-grained randomization schemes are similar to function-level randomization by utilizing different granularity location displacement or equivalent instruction substitution to invalidate the off-line analysis.

C. Memory Leakage

All the methods mentioned in section II-B increase the information entropy of randomness. Albeit with the existence of information disclosure vulnerability, drawbacks of some randomization techniques will be exposed. Information disclosure serves as auxiliary within an attack, with which the pointer errors are generated through the memory corruption bugs. Simultaneously, the memory layout is revealed with these pointers. Information disclosure is the most destructive techniques to bypass the randomization algorithms, which is exploited in many attacks, such as CVE-2013-2839[17] and CVE-2014-0322[18]. When the location of some code in memory is obtained by adversaries, the known memory space relevance can be used in the off-line analysis to speculate the actual memory layout, thus bypassing the protection of randomization.

In this paper, we only consider the memory leakage with which program vulnerabilities are exploited to read memory information through functions such as *Write()*. And information disclosure at the macro level, such as the disclosure of user identity information by side channel attack, is beyond the scope of this paper.

III. Assumption and Adversary Model

We now turn on our assumptions and adversary model. The adversary model is a theoretical foundation for section IV.

A. Assumption

This paper is concerned with attacks that bypass fine-grained randomization approaches, namely ones that give the attacker a leaked pointer in a function, then addresses of remaining functions are obtained by reading transfer instructions inside the function and then arbitrary malicious operations can be executed by reusing these code. Thus, we assume the adversary is able to exercise one of these pre-existing vulnerability and obtain the leaked pointer.

In what follows, we assume that the target platform uses the following mechanisms to mitigate the execution of malicious operations:

- **Non-Executable Memory:** We assume that the security model of non-executable memory is applied to the stack and heap. Hence, the adversary cannot inject code into the program's data area.
- **ASLR:** We assume that address space layout randomization is applied to the target platform.
- **Fine-Grained Randomization:** We assume that the target platform enforces fine-grained code randomization. In particular, we assume a strong fine-grained randomization scheme which permutes the order of functions [6].

Notice that, since ASLR is deployed, the first step of our adversary model is to bypass it with information leakage. In addition, we also assume that the adversary has the capability to analyze the vulnerable programs off-line.

B. Adversary Model

With these assumptions, we highlight our adversary model in Fig.1. In step 1, the adversary finds usable gadgets and the internal function dependency by off-line analysis. In step 2, given the leaked pointer, a particular function serving as the first ascertainable, readable and executable area of the attack model is located. More importantly, the following steps of attack model searching available gadgets and function call instructions start from the informations of this function. In step 3, from this function, the memory space is searched forwards and backwards by traversing the function call instructions in this function until the boundary of a function is identified. Methods searching for boundaries of functions in binary codes are categorized in two kinds. The first category is based on characteristics of function. For example, in x86, functions without optimization mostly begin with instructions like *push ebp; mov ebp esp; sub esp xxx;* and end with instructions like ret. Consequently, function space can be determined through features described above. Another is the heuristic method based on learning. Various scholars achieve their aim of identifying function space by training models with enormous features. By applying this searching technique iteratively on each function found, we can build a function dependency tree and collect the useful gadgets. In step 4, we construct the dispatcher with available

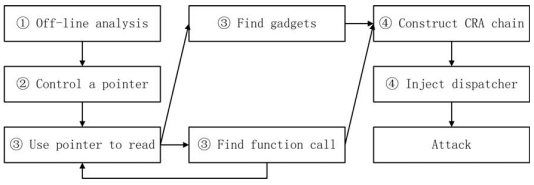

Figure 1. Adversarial Model

gadgets and inject it along with the invocation parameters into the target program's memory space.

This model collects not only the information of the gadgets but also the call relation between functions at the off-line analysis stage. It is the kernel on the demonstration that the function relationship (coupling) can be used by the adversary to acquire information about more gadgets.

Another famous way to bypass fine-grained ASR is Just-in-time ROP (JIT-ROP)[19]. It also requires a leaked pointer to trigger memory leaks, and the pointer is then applied to read pointed memory page code. Consequently, a number of memory pages are disassembled to find required gadgets by traversing the control flow instructions. But this method is not suitable for some fine-grained randomization methods. Modern compilers, such as LLVM, have already equipped each function with an exclusive section, page access errors is generated when traversing the contents of page space external to the function. Moreover, future randomization methods could conduct access control of finer granularity. More specifically, when using the current function, reading errors may occur over the upper and lower memory space beyond the range of the function. Therefore, the direct utilization of JIT-ROP is not necessarily successful. During every searching process, the attack model proposed in this paper searches for gadgets merely in the internal space of the determined function instead of moving pointers beyond the function space. Thus it can defend the defense strategy of *page-guard*.

IV. Security Metric of Function Risk

In this section, we give the function risk assessment model based on the adversary model proposed in Section III. It studies how function coupling would affect the capability of an adversary to finish the attack.

A. Function Risk Assessment

Security evaluation of function risk focusing on function coupling should be concentrated on whether adversaries can finish attacks with the leaked function pointer.

Here, we first give the overall security evaluation formula that studies what factors would affect the success of an attack:

$$Leak\_attack \cap Leak\_code \neq \emptyset \quad (1)$$

$$Leak\_attack = Attack\_plan_1 \cup ... \cup Attack\_plan_n \quad (2)$$

$$Leak\_code = Leak\_func_1 \cup ... \cup Leak\_func_n \quad (3)$$

Where, *Leak_attack* illustrates the possible attacking plans for an adversary to reach the goal; *Leak_code* represents all the gadgets available in the leaked functions; *Attack_plan* comprises the gadget chains that an attacker used to achieve the attacking purpose; *Leak_func* is a group of the gadgets the function leaked. We will introduce these definitions of the symbols in the follow-up in detail.

1) Attack_plan: There are a variety of ways for an adversary to accomplish the attack goal. For example, in the platform of x86, if adversaries want to use a system call, they can store the corresponding parameters in the register of eax and ecx, and then call `int 0x80` to complete the attack.

Adversary can also accomplish this goal by using the gadget *pop eax; pop ecx; ret*, with the only need to pre-deposit values of eax, ecx and addresses containing `int 0x80` instructions onto the stack. Besides, when adversaries can take control of the ebx register, they can exploit *mov ebx, eax; ret; mov ebx, ecx; ret* at the same goal. In addition, an adversary can exploit the dispatcher to complete the attack based on the JOP [20] attack and the COOP attack by modifying the pointer in the virtual function table. Since a variety of plans to achieve an attack are effective to adversaries, any successful plans can lead to severe security problems in software. We define all the aggregation of *Attack_plan* as *Leak_attack*, and it contains all possible plan that attacks can accomplish the goal.

2) Leak_func: Each *Leak_func* represents the gadgets adversaries can find by the leaked function pointer. These gadgets may be in different functions, but they can be found by function call graph. Through the program's vulnerability, an adversary may not merely locate a function address in the initial case, and there we use *Leak_code* to represent all gadgets that an adversary can get through information leakage. And *Leak_code* is a collection of all *Leak_func* gadgets.

As shown, there are various attack plans for an adversary to achieve the attacking goal. As long as the leaked code is sufficient to finish one of the attack plans, the attack can be considered to be theoretically successful. Therefore, the most important factor that affects the success of an attack is whether the *Leak_code* can be used to finish one attack plan. If it contains some high-risk functions, attackers may find many useful gadgets to accomplish the attack. In the next section, we will discuss how to measure the risk of a function.

B. Security Metric Model

It's universally known that the ROOT are capable of operating system unlimitedly. Therefore, the most direct and effective attacks of adversaries are conducted by obtaining system permissions, namely the ubiquitous $syscall(int\$80)$ in system library. Consequently, the functions accessible to $syscall$ with function correlation are defined as the highest risk function (Level 1), while the MHR, a bool value, serves as the criteria of whether the highest risk function is associated with $syscall$.

Generally, attackers are inaccessible to the location of $syscall$ and thus forced to accomplish attacks in a roundabout way by injecting particular malicious codes. Hence, adversaries tend to exploit the memory permission modification functions (*mprotect*, *virtualprotect*) or the executable memory application functions (*memalign*, *memset*) to guarantee the execution of their injected code. Eventually, functions accessible to the two category functions mentioned above are defined as the higher risk function (Level 2), while the HR is the criteria of whether a function can access to these functions.

Strictly speaking, without the level 1 nor the level 2 risk functions, the function itself is relatively vulnerable with combination of gadgets. Exploiting the indirect transfer instructions (e.g. call *%rax, jmp *%rax) for control flows' recombination, adversaries manage to transfer the execution flow to the location of desired gadgets. Therefore, functions that are accessible to indirect transfer instructions are defined as the medium risk function (Level 3), while the MR is the criteria of access to indirect transfer instructions.

Scarcely, adversaries are merely to modify certain data or register values in nondestructive purpose. Referring to functions that are accessible to low-risk function, for example, capable of partial arithmetical operation, we classified them as the low-risk function (Level 4). In consideration of CRA's proven Turing Completeness, all functional interval instructions are consequently regarded with potential computation behaviors and categorized in low-risk function. Especially, the LR is the criteria of access to arithmetic operations.

Here we introduce a four-dimensional axial vector to represent individually the risk level in sequence from high to low. Metric vector $\alpha_1 = (1, 0, 0, 0)$ represents for the highest risk level; $\alpha_2 = (0, 1, 0, 0)$ for the high risk level and so on. Following formulas illustrate the integrated risk of functions:

$$RISK\_FUNC = MHR{*}\alpha_1 + HR{*}\alpha_2 + MR{*}\alpha_3 + LR{*}\alpha_4 \quad (4)$$

According to the equation, the integrated risk of a particular function is obtained as RISK_FUNC, through which the existing risk of a given function can be intuitively judged. It can be inferred from the diversity of RISK value that 16 different risk value is possibly existed in a given function, specifically $(0, 0, 0, 0)$ $(1, 1, 1, 1)$. Simultaneously, program risk value is obtained by accumulating individual function risk value:

With the RISK_FUNC, the risk of a program can be defined as:

$$RISK\_PGM = \sum_{i=1}^{N} RISK\_FUNC_i \quad (5)$$

Table II
INFORMATION LEAKAGE FROM "MAIN"

Name	POP	MOV	ARI	Indirect call
Nginx	6794	55	3786	246
tar	54716	7848	21442	4168
omnetpp	58	79	10	17
soplex	1103	1861	244	37
bzip2	210	212	31	8
astar	287	232	125	1

With the $RISK\_PGM$ calculated above, risk value of a given program can be quantified. With each dimension representing exact number of risk categorized into corresponding risk level , the program risk of each dimension can be eventually evaluated, which is of critical importance to eliminate the program risk of each dimension.

C. Metric's Application

Both the adversary and defender could use this metric model.

For adversaries, when they are digging out memory leakage vulnerabilities, High-risk programs could be found according to the program risk metric model (or the value of *RISK_PGM*). Then, functions which are easily exploited to accomplish the attack target are shown with the function risk metric model (or the value of *RISK_FUNC*). Finally, the adversary could dig out vulnerabilities of those targeted functions.

There are two aspects that could use the metric model for defenders. First, our model can be used to measure the overall risk of a program, so the designers of security solutions can put forward lower risk program design or reinforcement plan to reduce the values of all dimensions of risk. On the other hand, defenders can find and fix vulnerabilities for higher risk function calculated by formula (4) to prevent adversaries from exploiting these functions to complete the attack target.

V. EXPERIMENT

In this section, we carry out the experimental analysis of the metric model mentioned above, with the open source software nginx, gzip to show the application of the metric model under Linux. These frequently used software are selected as ideal experimental tools for their compiling source code without obfuscation or packer featured as open source as well as convenience in static analysis.

A. Attack Model Experiment

Given the adversaries can locate the address of main function through information leakage, they can consequently acquire the number of gadgets from all kinds of ROP instructions, as shown in Table II. As we can see, an adversary can get thousands of available gadgets, which are very likely to meet the needs of one or several *attack_plan*.

Table III
STATISTICS OF ATTACK

Name	Function Number	Related to indirect call	registers of indirect call	Attack paths number
Nginx	1052	715	rax r13 r15 rdx rcx rbp	7039988
tar	790	718	rax r13 r15 r12 rdx rcx r8 r14 rbx rbp	56376638
omnetpp	2495	2249	rax rbx rcx r8 r12 rsi rdi	4767650
soplex	1499	266	rax rbx r12 rcx rdx	143188
bzip2	129	28	rax	16852
astar	161	6	rax	1386

We assumed that the adversary's purpose is to exploit the indirect call instruction to jump to any address. One *attack_plan* may need three instructions, the first of which is the indirect call instruction namely call **%rax*. In addition, as the *rax* registers are overwritten into the address the adversaries plan to jump to, adversaries need instructions to modify its value, such as *mov %rbx, %rax; RET* or *pop %rax; ret*. Therefore, we search for the program according to the target. In the experiment, we use static analysis to get the function that may accomplish the attack in assumption of that the adversary use only less than 3 gadgets to complete the *attack_plan*. Our statistical results are shown in Table III.

As shown, it's obvious that most mainstream software contains indirect function call instructions, among which *rax* is the most widely used register. Under the assumption that information disclosure vulnerability may exist in each function, with only a function vulnerability, registers are reachable to be controlled along enormous attack paths to launch an indirect call by our attack model. In different programs, the proportion of functions related to indirect jump instructions varies, as it is relevant to the extent of function coupling. With stronger coupling of functions, a single function may be directly or indirectly associated with more functions contained indirect jump instructions. In summary, some suggestions are given for software modification in section VI.

In order to automate the attack model, we have designed an automated attack process. Suppose there is a stack overflow vulnerability in the *compress* function in bzip2, that is, an adversary can control the execution stream by modifying the data in the stack. Then, adversaries can automatically attack along the path. It is shown in Fig. 2.

First of all, in the off-line analysis phase, we have learned that two functions related to compress function have instructions that can control the register of the indirect call instructions. Then, when conducting the actual attack, in the first step, we use the information disclosure vulnerability in the compress function. According to the correlation, we can find *uInt64_from_UInt32s* function, then we traverse the function space to find the stack pivot instructions to control the stack frame pointers pointing to our overflow

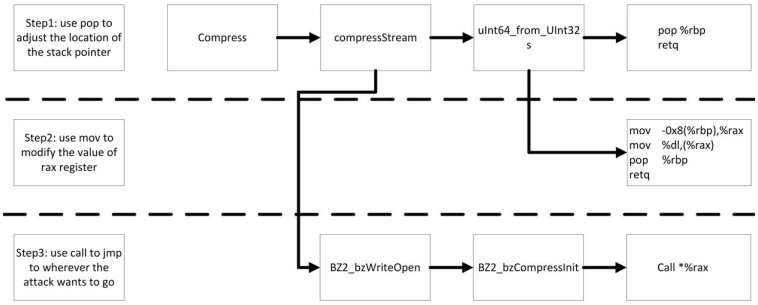

Figure 2. An example of attack path

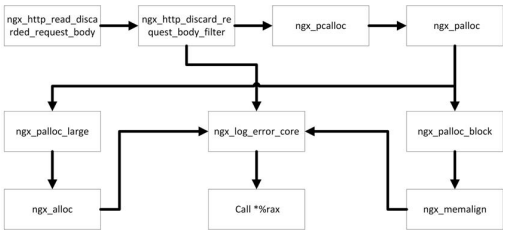

Figure 3. nginx attack paths

Table IV
SECURITY METRIC OF SHARED LIBRARIES

Name	Total LR	Total MHR	Total HR	Total MR	RISK_PGM
Libc.so	1741	1008	29	653	(1008,29,653,1741)
Libm.so	295	0	0	5	(0,0,5,295)
Ld.so	16	7	0	6	(7,0,6,16)
Libncurses.so	310	0	4	4	(0,4,4,310)
Libnsl.so	129	0	0	35	(0,0,35,129)
Libpthread.so	354	221	24	20	(221,24,20,354)
Libresolv.so	91	0	0	2	(0,0,2,91)

point. Then, in the second step, we continue to traverse the entire function and find the *mov, mov, pop,retq* instructions shown in the Fig. 2. The *rax* register is modified to the destination address that adversaries want to jump to with the instructions. Finally, we have found the *BZ2_bzCompressInit* function according to the correlation, and use the call instruction to finish the attack.

We also verify the feasibility of our attack model through a real vulnerability. CVE-2013-2058 is a Nginx buffer overflow vulnerability, caused by errors in integer types. This vulnerability occurs in *ngx_http_read_discarded_request_body*, where we start with this function and use our model to find that there are three attack paths that can invoke indirect function calls. The specific attack path is shown in Fig.3. It is known that adversaries can exploit functions with memory allocation or memory permission modification, namely *posix_memalign* and *mprotect*. According to experiments conducted with the attack model established, there are 500 functions along with 7446 attacking paths possibly lead to the adversary to locate the address of *posix_memalign*. Eventually, through proper parameters to invoke *posix_memalign*, adversaries can apply the executable permission to memory space.

B. Security Metric

In this section, some system libraries are analyzed according to the security metric model. Since the highest risk functions are generally found in shared libraries, library $libc-2.13.so$ is therefore analyzed subsequently. Given library possesses 1741 functions, of which 1008 are associated with $syscall$, while 5985955 paths can lead to $syscall$. Similarly, several other shared libraries are analyzed, the results of which are shown in Table IV.

The MHR, HR and MR represents the corresponding risk level of the whole library function. Through analysis of these library functions, there are two libraries which are accessible with *syscall* that are possibly exploited by adversaries. According to the value calculated by our model, adversaries tend to exploit libraries containing amount of *syscall* in attacks and locate function pointer leakage accessible to *syscall* with formula 4 accordingly. Additionally. when judging the severity (consisting of more high-risk instructions) of programs or libraries, we cannot rely merely on the quantity of code but the actual statistics that the accurate results can be obtained by our risk model.

VI. DISCUSSION

A. Advantages and Limitations

We establish an attack method to bypass randomization. The traditional evaluation of randomization only evaluates the randomization itself without the specific analysis of the application scenario. Our experimental results in this paper show that the effectiveness of the randomization depends not only on the randomization method, but also on the randomized program itself. On the other hand, compared with the general attack methods, the attack model established in this paper is closer to the realistic scene with fine-grained randomization, which can be used for automatic attacks.

There are some limitations for the model proposed in this paper. With the lack of the industrial deployment scenarios,

there are fewer examples to verify the model. And the adversary may use other types of gadgets which are not mentioned in this paper, so the model can be improved in the future.

B. Eliminating coupling

Through the analysis of the attack model and our experimental results, it can be concluded that it's the confirmation of functional relevance, namely the function coupling, that makes the adversaries locate other function and successfully complete attack by acquiring gadgets. Hence, in order to defend attacks relying on function coupling, defenders should be prudent on how to eliminating function coupling. In consideration of this, some solutions are given as follows:

1) Hide function call instructions in memory space invisible to adversaries, specifically in *sgx* or unreadable memory page, and subsequently replace these instructions with control flow transfer instructions in original program locations. Therefore, functional relevance is inaccessible to adversaries and memory leak is avoided more or less.
2) Insert some fake functional invoke and baffle adversaries from effective analysis. When fake functional invoke is pointed to particular locations that are likely to crash down programs, the actual coupling degree of functions are decreased and memory leak based on program internal correlation is defended.
3) Insert some fake control flow transfer instructions, and fabric an enclosure with unity of those transfer instructions through construction of transfer instructions' target location, with which analysis of adversaries conducted based on functional relevance would be invalid.

Solutions above can be realized with from source-code to binary-code and attributed to secondary reinforcement of existed programs. Although they can effectively alleviate attacks based on program internal relevance, we strongly recommend that program developer and program language designer can optimize their solutions at the stage of programing and compiling.

VII. Related Work

Modern Operating Systems (OS) use DEP to disable shellcode execution. Therefore, in order to bypass the data execution protection, Code Reuse Attack (CRA) uses the program's existing code(gadget) is generated. CRA includes ret2lib [21], ROP [3], and Jump-oriented programming (JOP) [20]. Nowadays, OS introduces the security mechanism of ASLR [4] to randomize the layout of memory segments, which increases the difficulty of the attacker on guessing the actual address. However, the attacker can bypass ASLR when they exploit the address leakage vulnerability to obtain reusable code memory address. For the ASLR limitations, the scholars proposed more granular randomization methods. For example, [16], [10], [15], respectively, they use function randomization, instruction randomization, and instruction equivalent replacement method to defend against CRA. However, the JIT-ROP [19] proposed by Snow et al. can bypass the these defenses by information disclosure and ROP attack. In the design of function-level randomization, Marlin randomizes the internal structure of the executable code by randomly shuffling the function blocks in the target binary [7]. They integrate Marlin into the bash shell that randomizes the target executable before launching it. Bin_FR [6] parses the binary directly, which do not rely on the source code, and implements function-level randomization and random padding in the code segment of the binary to shuffle the internal layout of the code segment.

The code reuse attack uses information disclosure to obtain the information needed for attack, such as memory layout speculation, memory out-of-bounds read and write, read object content, key address information acquisition and other techniques. And attackers use the key address information to construct the gadget chain to implement attack [22]. Protection schemes are always combined with randomization and memory page privilege protection. Readactor [23] defines the scope of controllable object access to prevent memory discloses across the boundary. XnR [24] uses read and execute mutex to prevent disclosure of critical address information, Giuffrida C et al. [8] proposes an approach to the operating system that minimizes the range of readability of the program, making the attacker's guessing ability lower, but these are fixed strategies and there is false code reading problem. The encryption of visible pointers and the runtime re-randomization [25] is a strong defense mechanism against both brute-force attacks and memory disclosure exploits. Kangjie Lu et al. present a novel mechanism, ASLR-Guard, which completely prevents the leaks of code pointers, and render other information leaks useless in deriving code address [26]. An alternative line of defense is to invalidate the leaked information or use destructive reads to prevent the execution of what was read [27]. Adrian Tang et al. present Heisenbyte, a system of thwarting memory disclosure attacks using destructive code reads [28]. In the development of adaptive and fine-grained strategy, proposing a more detailed information disclosure mitigation scheme can use the function risk model proposed in this paper. Therefore, in the presence of memory address information leakage case, the security metric of function randomization provides a new direction of exploration against code reuse attacks.

VIII. Conclusion

In this paper, we have established an attack model with leakage information of certain given code pointer based on attack features, which can bypass the fine-grained randomization. The function risk is quantified for the first time by judging whether the attacks are completed through leakage of the function pointer. Our experimental results have shown

that enormous individual function leakage will lead to effective attacks to specific targets. Hence, we propose that functions' coupling should be eliminated by programmers to alleviate the threat resulted by information disclosure. As for the fine-grained randomization, our experimental results have illustrated that the judgment criteria depends on the randomization methods and moreover on the randomized program itself. Therefore, we suggest that future randomization evaluation should be the main basis of judging whether codes can be hidden or not rather than the transfer distance of the code.

ACKNOWLEDGEMENTS

Partly supported by the National Natural Science Foundation of China(61373168, U1636107), and Doctoral Fund of Ministry of Education of China (20120141110002).

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Eternal war in memory," in *Security and Privacy*, 2013, pp. 48–62.

[2] N. Trev, "Data execution prevention," *Lect Publishing*, 2011.

[3] E. Buchanan, R. Roemer, S. Savage, and H. Shacham, "Return-oriented programming: Exploitation without code injection," *Black Hat*, 2008.

[4] P. Team, "Pax address space layout randomization (aslr)," 2003.

[5] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," 2002.

[6] J. Fu, X. Zhang, and Y. Lin, "Code reuse attack mitigation based on function randomization without symbol table," in *Trustcom*, 2016, pp. 394–401.

[7] A. Gupta, J. Habibi, M. S. Kirkpatrick, and E. Bertino, "Marlin: Mitigating code reuse attacks using code randomization," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 1–1, 2015.

[8] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. of the Usenix Security Symp*, 2012, p. 40.

[9] J. Fu, R. Jin, and Y. Lin, "Frprotector: Defeating control flow hijacking through function-level randomization and transfer protection," in *Proceedings of the 13th International Conference on Security and Privacy in Communications Networks, SecureComm*, 2017.

[10] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring:self-randomizing instruction addresses of legacy x86 binary code," in *ACM Conference on Computer and Communications Security*, 2012, pp. 157–168.

[11] D. Xu, J. Ming, and D. Wu, *Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method*. Springer International Publishing, 2016.

[12] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm – software protection for the masses," in *Ieee/acm International Workshop on Software Protection*, 2015, pp. 3–9.

[13] Y. Liang, X. Ma, D. Wu, X. Tang, D. Gao, G. Peng, C. Jia, and H. Zhang, *Stack Layout Randomization with Minimal Rewriting of Android Binaries*. Springer International Publishing, 2015.

[14] S. H. Kim, L. Xu, Z. Liu, Z. Lin, W. W. Ro, and W. Shi, "Enhancing software dependability and security with hardware supported instruction address space randomization," in *Ieee/ifip International Conference on Dependable Systems and Networks*, 2015, pp. 251–262.

[15] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[16] J. Hiser, A. Nguyen-Tuong, M. Co, and M. Hall, "Ilr: Where'd my gadgets go?" in *Security and Privacy*, 2012, pp. 571–585.

[17] "Cve-2013-2839," https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2839.

[18] "Cve-2014-0322," https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0322.

[19] K. Z. Snow, F. Monrose, L. Davi, and A. Dmitrienko, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy*, 2013, pp. 574–588.

[20] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March*, 2011, pp. 30–40.

[21] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[22] F. Jianming, L. Xiuwen, T. Yi, and L. Pengwei, "Survey of memory address leakage and its defense," *Jounral of Computer Research and Development*, vol. 53, no. 8, pp. 1829–1849, 2016.

[23] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Security and Privacy*, 2015, pp. 763–780.

[24] M. Backes, T. Holz, B. Kollenda, P. Koppe, and J. Pewny, "You can run but you can't read:preventing disclosure exploits in executable code," pp. 1342–1353, 2014.

[25] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *The ACM Sigsac Conference*, 2015, pp. 268–279.

[26] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *ACM Conference on Computer and Communications Security*, 2015, pp. 280–291.

[27] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 35–46.

[28] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *ACM Sigsac Conference on Computer and Communications Security*, 2015, pp. 256–267.